Model Variables: Cleanly Supporting Abstraction in Design By Contract

Yoonsik Cheon, Gary T. Leavens,
Murali Sitaraman, and Stephen Edwards

TR #03-10b
April 2003; revised September 2003 and August 2004.

Keywords: documentation, formal methods, model variables, programming by contract, runtime assertion checking, specification languages, tools, Java language, JML language.

2001 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, validation, JML; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, mechanical verification, pre- and post-conditions, specification techniques.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Model Variables: Cleanly Supporting Abstraction in Design By Contract

Yoonsik Cheon
Dept. of Computer Science
University of Texas at El Paso
El Paso, TX
`cheon@cs.utep.edu`

Gary T. Leavens
Dept. of Computer Science
Iowa State University
Ames, IA
`leavens@cs.iastate.edu`

Murali Sitaraman
Dept. of Computer Science
Clemson University
Clemson, SC
`murali@cs.clemson.edu`

Stephen Edwards
Dept. of Computer Science
Virginia Tech
Blacksburg, VA
`edwards@cs.vt.edu`

Abstract

In design by contract (DBC), assertions are typically written using program variables and query methods. The lack of separation between program code and assertions is confusing, because readers do not know what code is intended for use in the program and what code is only intended for specification purposes. This lack of separation also creates a potential runtime performance penalty, even when runtime assertion checks are disabled, due to both the increased memory footprint of the program and the execution of code maintaining that part of the program's state intended for use in specifications. To solve these problems, we present a new way of writing and checking DBC assertions without directly referring to concrete program states, using "model", i.e., specification-only, variables and methods. The use of model variables and methods does not incur the problems mentioned above, but it also allow one to write more easily assertions that are abstract, concise, and independent of representation details, and hence more readable and maintainable. We implemented these features in the runtime assertion checker for the Java Modeling Language (JML), but the approach could also be implemented in other DBC tools.

1 Introduction

Design by contract (DBC) is useful for checking the correctness of a program with respect to its specification [19, 20, 21]. In DBC, a class and its clients have a contract with each other. The client must guarantee, before calling a method m implemented by the class, that m's *precondition* holds, and the class must guarantee that m's *postcondition* holds after such a call. The *predicates* or *assertions* found in pre- and postconditions are usually

```
public abstract class SortedIntListApprox {
  public int size = 0;

  //@ ensures size == \old(size + 1);
  public void add(int elem) { size++; /* ... */ }
}
```

Figure 1: The helper variable approach. In the JML notation used in this and subsequent examples, a method's specification precedes its signature, and *annotations* are contained in comments that start with at-signs (`@`). As in Eiffel, `\old(`e`)` means the value of e in the pre-state.

written in a form that can be compiled, so that violations of the contract can be detected at runtime.

It is important that the precondition of a method m should not directly mention program fields, methods, and types that have less visibility than m, because doing so would make it impossible for clients to determine whether a call to m is correct [21, pp. 357-9]. This rule is enforced by Eiffel [20, Sec. 9.8]. Unfortunately, Eiffel and several other DBC tools allow private features to appear in postconditions for public methods [20, 21]. Doing so leads to coupling between assertions and implementation details, making it difficult for clients to understand assertions. To avoid these coupling and understanding problems, the following conventional approaches have been proposed to avoid using private details in public specifications [22]:

- *Helper variables.* This approach introduces additional variables (i.e., fields) to help specify state changes. For example, in Figure 1 the `add()` method of a sorted list type is specified in terms of a helper field, `size`, that counts the number of elements in the list.

- *Query methods.* This approach introduces query methods to help write assertions. A *query method* is a side-effect-free method that observes the program's state. For example, Figure 2 uses two query methods, `contains()` and `size()`, to specify the behavior of `add()`.

- *Immutable types.* This approach introduces new types, with immutable objects, to store the (abstract) state of an object; the behavior of each method is then specified in terms of this state. Figure 3 uses the type `JMLEqualsSequence` to specify the abstract state of a sorted list object.

The immutable type approach cannot be used in isolation, but must always be combined with either the helper variable or query method approaches. This is beneficial, since both the helper variable and the query method approaches tend to underspecification, and supplementing these approaches with the immutable type approach often helps fix such underspecification problems. For example, the specification for `add()` in Figure 1 does not constrain the contents of the list in its post-state. An implementation that satisfies this specification might insert a random integer (instead of `elem`), and would not have

```
public abstract class SortedIntListQuery {
  /** Is the given element in this list? */
  public abstract boolean contains(int elem);

  /** Return the size of this list. */
  public abstract int size();

  //@ ensures contains(elem) && size() == \old(size() + 1);
  public void add(int elem) { /* ... */ }
}
```

Figure 2: The query method approach.

```
import org.jmlspecs.models.JMLEqualsSequence;
public abstract class SortedIntListImmutType {
  public JMLEqualsSequence theList = new JMLEqualsSequence();

  /*@ public invariant theList != null
    @  && (\forall int i; 0 <= i && i < theList.size();
    @            theList.itemAt(i) instanceof Integer)
    @  && (\forall int i; 0 < i && i < theList.size();
    @            ((Integer)theList.itemAt(i - 1))
    @            .compareTo(theList.itemAt(i)) <= 0);
    @*/

  /*@ ensures theList.isInsertionInto(\old(theList),
    @                                  new Integer(elem));
    @*/
  public void add(int elem) {
    /* ... */
    theList = /* ... recompute theList's value ... */;
  }
}
```

Figure 3: The immutable type approach. This example uses the type `JMLEqualsSequence`, whose objects are immutable. It also uses an invariant and quantifiers. Quantifiers in JML start with `\forall` or `\exists` and are followed by a declaration; after the declaration is an optional range predicate, which is followed by a body predicate. Many quantifiers with finite ranges, such as the ones in this example, can be executed by JML's runtime assertion checker.

to leave the list in sorted order. While the query method approach can be used to write complete specifications, one has to use some discipline, such as that described by Guttag and Horning [7], to avoid underspecification. Indeed, Figure 2 also would allow an implementation that inserts random integers (this can happen, for example, if `contains()` always returns true) or one that does not sort the list. Our experience has shown that the easiest way to avoid these underspecification problems is to use the immutable type approach, as illustrated in Figure 3.

The helper variable approach is simple and direct, but in standard DBC tools it has two additional problems. The first problem is that if the introduced field is public, then it is unsafe to allow clients to change it. For example, in Figure 1, clients must not change `size`.[1] The second problem with the helper variable approach is that languages like C# and Java do not allow non-constant fields to be declared in interfaces. Thus, one has to write specifications for interfaces using the query method approach. For example, consider using a Java interface instead of an abstract class in Figure 3. In an interface, the field `theList` could not be declared. Thus, uses of `theList` in assertions would have to be replaced by uses of an abstract method that returns its intended value. Furthermore, because interfaces do not contain constructors, even using the query method approach does not allow one to describe the initial values of an interface's objects.

The three approaches summarized here suffer from software engineering and performance problems. While all of them attempt to avoid implementation details in specifications, they remain deficient because the introduced helpers become regular program fields, methods, and types even though they are intended only for writing assertions. As a result, the roles of fields, methods, and types become ambiguous.

Further, because there is no declaration of what public features are intended purely for specification purposes, these helpers are open to use in client code just like all other public fields, methods, and types. Public fields are a special cause for concern, since clients can use them to break invariants and cause other uncontrolled changes to the abstract state of objects. Together these problems compromise ease of software understanding and maintenance.

The lack of any distinction between code intended for specification and implementation leads to a performance penalty as well, even when runtime assertion checks are disabled. This is because helper fields are still maintained and manipulated. For example, it is easy to see in Figures 1 and 3 that the helper fields are manipulated in the bodies of methods such as `add()`. However, even in Figure 2 `add()` must update some storage to track the list's size. Since helpers might be used by clients, they are difficult to remove via conditional compilation or other approaches when assertion checks are disabled.

2 Model Variable Approach

To overcome the software engineering and performance problems of previous approaches, we explain how to extend DBC languages with specification-only declarations. In this paper, we limit ourselves to the simplest kinds of specification-only declarations—model variables and model methods. The qualifier "model" is used to distinguish declarations

[1]Java, and similar languages, could make this automatic if they allowed a class to declare fields that its clients could only read, but not write.

that are intended only for specification. We also limit our attention to languages such as C# and Java, where the main kind of variable is a field. Therefore, in the remainder of this paper we only discuss model fields. A *model field* is a specification-only field that describes the abstract state of some program fields [15]. A *model method* is a specification-only method [13].

Model fields and model methods are recognized as specification-only declarations by DBC tools, avoiding code-specification confusion. Since the assertions are written with model fields that are independent of representation details, our experience shows that they are easier to read and modify.

Specification-only assertions can also be manipulated more easily in formal reasoning and runtime assertion checking. A class that implements a specification with model fields can specify *abstraction functions*, which say how to map from values in the class's program fields to the "abstract values" in model fields [10, 17]. If an abstraction function is executable, then assertions written using model fields become executable, and can be used in runtime assertion checking. If convenient, abstraction functions can be specified using model methods. Similarly, if a model method m has a body, then calls to m become executable, and can be used in runtime assertion checking. Notably, the use of executable abstraction functions also eliminates the need for methods like `add()` to continually update the state stored in immutable variables (Figure 3).

It is not necessary to abandon the traditional approaches of helper variables, query methods, and immutable types to use model fields and model methods in DBC. Indeed, model fields can be used with the helper variable approach, since they can be made public without the danger of being changed by client program code. Furthermore, model variables can be combined with the immutable type approach to yield all the benefits of that approach without the corresponding ambiguity of purposes.

We have implemented model fields and methods in JML's runtime assertion checker [5]. JML is a formal interface specification language for sequential Java; it has sophisticated features for writing abstract, precise, and complete behavioral descriptions of Java classes and interfaces [13, 14]. However, it can also be used as a DBC language. The JML runtime assertion checker generates Java bytecode from JML annotated Java code. As is usual in DBC tools, unless an assertion is violated, assertion checking is transparent; except for performance measures (time and space), the behavior of the original program is unchanged.

It is also possible to mark program elements as usable both in program code and in specifications. This is the idea behind the `spec_public` and `pure` modifiers in the Java Modeling Language (JML) [13]. The `spec_public` modifier says that a non-public field is public for specification purposes; this has the effect of making the field be read-only by clients for specification purposes. For example, the `size` field in Figure 1 would be better if declared `private` and `spec_public`. The `pure` modifier says that a method has no side effects, which allows it to be used in assertions in JML (as in Figure 2). These annotations make it clear that such program elements are playing two roles, whereas `model` elements can only be used in assertions. On the other hand, private fields that are not `spec_public` cannot be used in public JML specifications; furthermore, program methods that are not pure cannot be used in assertions. Hence, JML allows one to clearly state the purpose of a field or method.

While we use JML to explain our approach, the ideas could be implemented in other interface specification languages or DBC tools. The contribution of this paper is a description of a language-independent idea and its implementation: namely, specification-only declarations.

2.1 Interface Specification Example

To illustrate the use of specification-only declarations, Figure 4 shows the Java interface `SortedIntListType`. This interface is specified in JML using a combination of the helper variable and the immutable type approaches, facilitated by a model field. As an interface, it can be implemented by several different classes, using arrays or binary search trees as their data structures. The specification starts with an annotation that does a `model import` of the types in `org.jmlspecs.models`; these are immutable types, like `JMLEqualsSequence`, that JML provides for describing abstract values in the immutable types approach. Since this is a model import in an annotation, the import is not part of the program as seen by a traditional Java compiler.

The key feature of this example is found in the next annotation, which declares a model field. This field, `theList`, contains the abstract value of a sorted list, and cannot be used in the body of a normal Java method, such as `add()`. The modifier `instance` in this declaration is used to state that `theList` is a non-static field, because Java fields declared in an interface are static by default. Therefore, each object of this interface has its own separate storage for the model field. The modifier `non_null` is a shorthand way of saying that `theList` is never null. The `initially` clause specifies constraints on the initial value of `theList` [23].

The model field `theList` is intended to be a sorted sequence of integers, with the values in ascending order. The invariant specifies the details of this model. It says that, all objects in the sequence are instances of the type `Integer` (and thus not null), and that the sequence is sorted in ascending order. JML enforces information hiding by ensuring that public invariants can only mention publicly visible names.

The model field is used to write the specifications of the methods `size()`, `get()`, and `contains()`. All of these methods are declared to be `pure`, which says that they cannot have side effects. This allows them to be used in the specifications as query methods. To see how using a model helps in writing more complete specifications, compare the specifications of `size()` and `contains()` in Figure 4 with those in Figure 2, where, using the query method approach, `size()` and `contains()` have no postconditions of their own. The specification of the `get()` method demonstrates a precondition, which is given by a `requires` clause in JML.

The specification of the method `add()` also uses the model, where the pure method `isInsertionInto()` is provided by the immutable type `JMLEqualsSequence`. Note, however, that the model field `theList` does not have to be maintained by the code the programmer writes; this is an advantage versus the helper variable approach. The postcondition does not mention that the resulting list is in sorted order, because that has already been stated in the invariant.

```
//@ model import org.jmlspecs.models.JMLEqualsSequence;
public interface SortedIntListType {
  /*@ public model instance non_null JMLEqualsSequence theList;
    @ public initially theList.isEmpty();
    @ public instance invariant
    @   (\forall int i; 0 <= i && i < theList.size();
    @         theList.itemAt(i) instanceof Integer)
    @  && (\forall int i; 0 < i && i < theList.size();
    @         ((Integer)theList.itemAt((int)(i - 1)))
    @         .compareTo(theList.itemAt(i)) <= 0);
    @*/

  //@ ensures \result == theList.length();
  /*@ pure @*/ int size();

  /*@ requires 0 <= index && index < size();
    @ ensures \result
    @   == ((Integer)theList.itemAt(index)).intValue();
    @*/
  /*@ pure @*/ int get(int index);

  //@ ensures \result == theList.has(new Integer(elem));
  /*@ pure @*/ boolean contains(int elem);

  /*@ ensures theList.isInsertionInto(\old(theList),
    @                                 new Integer(elem));
    @*/
  void add(int elem);
}
```

Figure 4: An interface specification with model fields.

2.2 Implementation Example

To give an example of an abstraction function, consider Figure 5. In this figure, the class `SortedIntList` implements the `SortedIntListType` interface from Figure 4. The code in Figure 5 uses four private fields to form binary search trees. The `isEmpty` field tells whether the other fields are defined. As the invariant states, `isEmpty` is true if and only if the left and right subtrees are null.

The abstraction function is specified in the `represents` clause of Figure 5. It says that the value of the model field `theList` is determined by the expression `abstractValue()` that appears on the right hand side of the represents clause. Following this clause, the model method `abstractValue()` is given. As a model method, it can only be called from within specifications. The code for this method can access types that were imported by model imports. Otherwise, its body is a regular Java method body that uses the type `JMLEqualsSequence`. JML's runtime assertion checker can execute model methods that have bodies. Thus, given an object of type `SortedIntList`, the runtime assertion checker can compute the value of the model field `theList` using this abstraction function. This approach, in turn, enables the execution of the other assertions in `SortedIntList` and those inherited from the `SortedIntListType` interface, such as the invariant and the postcondition of `add()`.

2.3 Summary of the Approach

Using model features in specifications fixes the problems we noted earlier in DBC. Because a model field or method is not a program feature, it does not confuse readers about its intended use. The declaration itself clearly reveals its intended role. Model features cause no runtime penalty, in either time or space, when runtime assertion checks are turned off; with checks disabled, model features are simply not compiled into bytecode by the JML compiler. The use of model fields in particular facilitates maintenance. A change in a type's private fields does not require a change in its client-visible specification, because specifications are written in terms of its model fields (and other public features). Changes to private fields thus only affect represents clauses (i.e., definitions of abstraction functions). In JML, represents clauses must be private if they refer to private fields (as in Figure 5), so such a change will not affect any prior reasoning about the specification done by clients. Finally, using model fields allows one to write specifications for Java interfaces in a model-oriented style.

3 Implementation of Model Features

This section explains the implementation of model methods and model fields in the JML compiler [5].

3.1 Model Methods

The JML compiler translates each model method with a body, like `abstractValue()` in Figure 5, into bytecode for a Java method. This allows model methods to be called when checking assertions.

```
//@ model import org.jmlspecs.models.JMLEqualsSequence;
public class SortedIntList implements SortedIntListType {

  private boolean isEmpty;
  private int val;
  private SortedIntList left, right;

  //@ private invariant isEmpty == (left == null && right == null);

  //@ private represents theList <- abstractValue();
  /*@ private model pure JMLEqualsSequence abstractValue() {
    @   JMLEqualsSequence ret = new JMLEqualsSequence();
    @   if (!isEmpty) {
    @       ret = left.abstractValue()
    @               .insertBack(new Integer(val))
    @               .concat(right.abstractValue());
    @   }
    @   return ret;
    @ }
    @*/

  //@ ensures theList.isEmpty();
  public SortedIntList() { isEmpty = true; }

  /* ... */

  public void add(int elem) {
    if (isEmpty) {
        isEmpty = false;  val = elem;
        left = new SortedIntList();
        right = new SortedIntList();
    } else {
        if (elem <= val) { left.add(elem); }
        else { right.add(elem); }
    }
  }
}
```

Figure 5: An example implementation with JML-annotated Java code.

```
public JMLEqualsSequence model$theList$SortedIntListType() {
  JMLEqualsSequence rac$v0 = null;
  rac$v0 = this.abstractValue();
  return rac$v0;
}
```

Figure 6: Code generated for `theList`'s access method in the concrete class `SortedIntList`.

When compiling a call to a model method, the main complication has to do with support for separate compilation. Separate compilation implies that some classes or interfaces may not be compiled with the JML compiler, and hence the bytecode for those model methods may not be available at runtime. To make calls to potentially unavailable methods work, the compiled code calls model methods via Java's reflection facility [5, Chapter 7]. Otherwise the treatment of model methods calls is relatively straightforward.

3.2 Model Fields Overview

The JML compiler does not allocate storage for model fields, and instead the value of a model field is determined by calling an *access method*—a generated method that returns the field's value. Each reference to a model field is thus compiled into a call to its access method.

The access method for a model field, x, encapsulates the abstraction function for x, given in x's represents clause. For example, the JML compiler generates the access method shown in Figure 6 for the represents clause of the model field `theList` (from Figure 5). The body of the model field access method in Figure 6 simply evaluates the expression part of the represents clause and returns the resulting value. The example also shows that the runtime assertion checker can evaluate an abstraction function written in terms of model methods (e.g, `abstractValue()`). If the abstraction function were defined in terms of other model fields, then calls to their access methods would appear in the body of the access method in the same way. This allows one to write assertions using layers of abstractions.

If a type does not inherit or define an abstraction function for a model field x, then the compiler generates a default implementation for its access method. An example is shown in Figure 7 for the model field `theList` declared in the interface `SortedIntListType` (from Figure 4). In Figure 7, the default access method's body makes an indirect downcall, using techniques which we explain below, that searches for an overriding implementation of the model field's access method. If no such overriding implementation is provided, for example, in a class implementing `SortedIntListType` in which there is no functional represents clause for `theList`, then this method throws a pre-defined exception to indicate an occurrence of a non-executable specification construct. (The runtime assertion checker in JML catches these exceptions, and interprets the smallest, enclosing boolean expression as being either true or false, depending on the context [5, Chapter 3].)

The indirect interpretation of model field accesses by assertions is one of the main complications in implementing model fields. That is, even though specifications in the

```
public JMLEqualsSequence model$theList$SortedIntListType() {
  java.lang.String cn = self.getClass().getName();
  java.lang.Object obj
     = rac$invoke(cn, self,
                  "model$theList$SortedIntListType",
                  null, null);
  return ((JMLEqualsSequence)obj);
}
```

Figure 7: Code generated for `theList`'s access method in the interface `SortedIntListType`.

`SortedIntListType` interface refer to the model field `theList`, the meaning of this model field is determined by a represents clause that appears in some concrete class, such as `SortedIntList` that implements the interface. Typically each different implementation of the `SortedIntListType` interface provides a different represent clause for the model field. Indeed, this will be the case whenever an interface or abstract class declares a model field, x, but does not declare program fields to represent x; each concrete class would then have its own abstraction function mapping the state of its concrete fields to the abstract value of x. Hence it is necessary to dispatch to the right access method at runtime. This dispatching process is especially complex for interfaces.

3.3 Interface Model Fields and Surrogate Classes

A model field declared in an interface, as in Figure 4, is called an *interface model field.* Two problems with interface model fields are where their access methods are defined (in the compiled output) and how these access methods refer to the program state.

The compilation of an interface in JML places access methods in a static inner class, called a *surrogate class.* The compiler cannot add assertion checking methods directly to an interface because in Java all interface methods must be abstract. To work around this, the compiler generates a surrogate class, as a static inner class of the interface, for each interface it compiles. This surrogate class hosts the methods that are needed for runtime assertion checking; these methods include both model field access methods, and other methods, such as methods that encapsulate pre- and postconditions.

To access the concrete state of an object that implements the interface, the methods of the surrogate class need to call the proper overrides of model field access methods. These overriding methods are found, using Java's reflection mechanism in a way described in the next subsection, via a field (i.e., `self` in Figure 7) in the surrogate class that references the current object being checked. This field is initialized to the current object; the constructor of the surrogate class takes the current object (the one implementing the interface) as an argument. The runtime assertion checker also maintains a mapping from the current object to its surrogate objects to allow dynamic calls from the current object to the interface's assertion checking methods and model field access methods. The references from the surrogate object back to the current object allow calls to the model field access methods from the surrogate object's methods to find the overrides needed to access the concrete program state in the current object. In sum, the responsibilities for both assertion checking and access to model fields are distributed over the object being checked and various surrogate objects for interfaces.

3.4 Dispatch for (Multiple) Inheritance

JML supports multiple inheritance of specifications, since a class inherits specifications from its superclasses and all the interfaces that it implements. Similarly, an interface inherits specifications from all its superinterfaces. Since Java only supports single inheritance, this raises a problem. How can multiple inheritance, and in particular overriding of multiply inherited model field access methods, be accomplished?

To illustrate how multiple inheritance is accomplished in JML, we first consider a simple case, where an implementing class provides an abstraction function for an inherited interface model field. This is the case for the class `SortedIntList`. The access methods of the field `theList` for the class `SortedIntList` and the interface `SortedIntListType` are shown in Figure 6 and 7 respectively. How are these access methods called to support inheritance of the model field `theList`? In particular, how is an inherited assertion written in terms of `theList`, e.g., the postcondition of the `add()` method, interpreted with the abstraction function provided by `SortedIntList`? Figure 8 answers these questions by showing the sequence of internal method calls that happen when a client calls the `add()` method on an instance of `SortedIntList`. Essentially two dynamic calls are made in opposite directions. The first dynamic call occurs from the object being checked to its surrogate object to evaluate the inherited postcondition of `add()`. The occurrence of `theList` in the `add()`'s postcondition leads to a call to the model field access method of `SortedIntListType` that in turn triggers the second dynamic call. The second dynamic call occurs in the opposite direction and retrieves the abstract value of `theList` from the current program state. Thus, the inherited assertion is evaluated in the proper state using the appropriate abstraction function.

The general situation is more complicated than the one illustrated by the simple example here. A subinterface may specify an abstraction function for an inherited interface model field (e.g., in terms of other model fields). That abstraction function must be used to interpret any occurrence of the model field, both in the superinterface and the implementing class. Our approach is to code method dispatch explicitly by making the assertion checking methods play the role of either a delegation or dispatch method (see Figure 9) [5, Chapter 7]. A model field access method in an interface where no abstraction function is defined plays the *delegation role*, since it delegates to the current object implementing the interface. A model field access method in a class with no abstraction function defined for that field plays a *dispatch role*, when it dispatches to some interface that does define an abstraction function for a model field it inherits. An example is `model$v$I`$_1$`()` of the class S of Figure 9. This dispatch method calls the correct model field access method (e.g., `model$v$I`$_1$`()` of the interface I_2), if it exists. Of course, the invoked access method is to be generated from an interface's represents clause and added to the surrogate class of the interface. As the dispatch method is generated for an implementing class, it is possible to determine statically whether an applicable interface represents clause exists. If there is no such represents clause in the interface, then the compiler issues a warning message and at runtime the dispatch method throws a pre-defined exception to indicate non-executability.

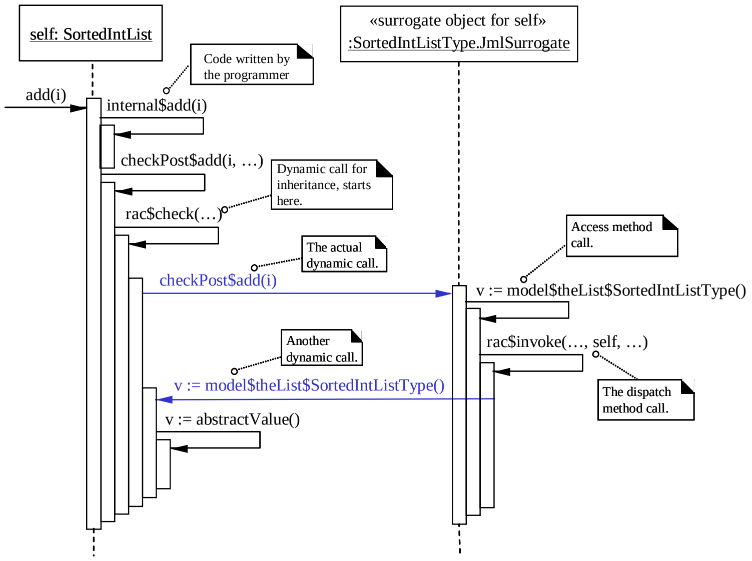
self: SortedIntList
«surrogate object for self»
:SortedIntListType.JmlSurrogate
Code written by the programmer
add(i)
internal$add(i)
checkPost$add(i, …)
Dynamic call for inheritance, starts here.
rac$check(…)
The actual dynamic call.
Access method call.
checkPost$add(i)
v := model$theList$SortedIntListType()
rac$invoke(…, self, …)
Another dynamic call.
The dispatch method call.
v := model$theList$SortedIntListType()
v := abstractValue()

Figure 8: Dynamic calling of model field access methods.

This default dispatch method may be overridden by a subclass, i.e., if a subclass provides an abstraction function for the model field.

Figure 9 shows an example of the more general case of specification inheritance. The class S inherits a model field, v, from the interface I_1 and a represents clause from the subinterface I_2. The use of the model field v in the interface I_1—e.g., in the precondition, $P(v)$, of the method $m()$—is translated into a call to the model field access method, which in this case forwards the call to the dispatch method added to the implementing class S. In turn the dispatch method calls the appropriate access method, i.e., the one added to the subinterface I_2. Thus, the use of the model field v in the interface I_1 is appropriately interpreted by objects of the class S by calling the abstraction function given in the subinterface I_2.

4 Related Work

The use of abstract values in specification is not new, as shown by the immutable type approach described in the introduction. Furthermore the use of abstract values has been a feature of several formal specification languages.

Anna [18], an annotation language for Ada, makes a distinction between program code and specification-only declarations with its “virtual text” mechanism. Besides variables, it is possible to declare procedures and types in virtual text; JML allows similar declarations, but does not currently compile types declared in this way for runtime assertion checking. As in JML, variables declared as virtual text in Anna can be used only in annotations, e.g., in assertions. However, the main difference is that variables declared in virtual text in Anna are explicitly assigned to by statements in virtual text. JML has a similar feature,

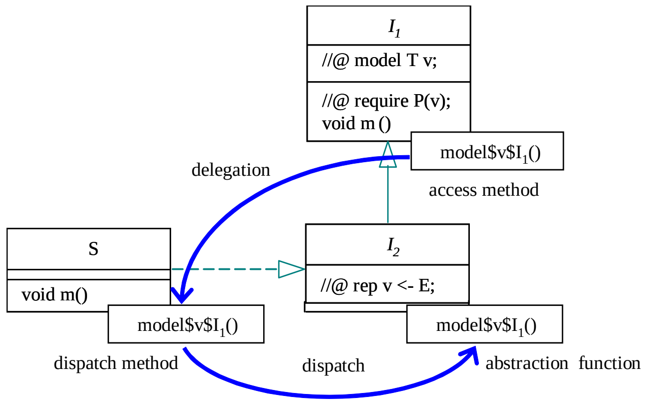

Figure 9: Inheritance of interface model fields and represents clauses.

called "ghost variables" and "set statements" that appear only in JML annotations that manipulate them. However, Anna does not have the equivalent of JML's model variables, which are specification-only variables whose values are implicitly given by user-declared abstraction functions. Anna also does not have the equivalent of JML's `spec_public` modifier.

In Larch family of interface specification languages [8], specifications are written solely in terms of abstract values, specified algebraically, and no explicit mapping is specified between abstract and representation values.[2] What distinguishes our approach from work on Larch is the use of specification-purpose declarations to hold abstract values that can be used in runtime assertion checking. This is primarily due to the explicit specification of abstraction functions in JML.

The RESOLVE family [23] is similar to the Larch family in that it also is a family of interface specification languages. Unlike most Larch-style languages, however, RESOLVE features a way to specify abstraction functions. Hegazy has discussed the use of abstraction and mathematical models in component-based software testing [9]. More recent work by Edwards *et al.* [6] shares our objective of employing abstraction in assertion checking, using model-based RESOLVE specifications [26] and C++ implementations. Their assertion checking approach uses abstraction functions (and in general, abstraction relations) that are given in implementations, as we do in our approach. However, the focus of this previous work has been on automatically generating wrappers to check assertions. The code for checking the assertions must be supplied by programmers and the underlying tool merely incorporates them at appropriate places. We present a more comprehensive approach and a Java-based tool in this paper whereby much of the assertion checking is automated. Combining the idea of wrappers with the JML checker is part of our current research.

The Abstract State Machine Language (AsmL) [1] supports executable specifications written using abstract variables. However, the abstract variables in AsmL are not associated with program variables through abstraction functions directly (although this can be done). Instead, most AsmL specifications are written as abstract programs that ma-

[2]However, Larch/C++ also has some of the features found in JML [12].

nipulate abstract variables directly. Instead of constraining program code, as is done in DBC, AsmL specifications are usually kept separate from program code. Specifications are normally executed separately from programs when doing runtime assertion checking, and violations are detected by comparing the two outputs [2]. This separation of specifications from programs is thus a paradigm shift from DBC.

In Eiffel [20, 21], there are no specification-only declarations. Partly based on our work, Schoeller [25] recently proposed to introduce a set of immutable model types for the purpose of writing contracts. Eiffel's design has been influential in the DBC community; for example, aside from JML, other design by contract tools for Java, such as iContract [11] and Jass [3], have no explicit support for abstract values. As a result, Eiffel-based approaches make it more difficult to write abstract specifications and keep them distinct from code.

For specification-only fields, JML builds on the work of Leino and his co-authors [15, 16]. Leino's work clarified the semantics of specification-only variables, particularly with respect to frame axioms [4] (i.e., `modifies` or `assignable` clauses). Leino introduced specification-only variables to solve the problem of information hiding while still being able to specify and verify programs in a model-oriented style. However, Leino and his co-authors have not employed these ideas in runtime assertion checking or DBC tools. Their focus is on connecting specification-only variables to the program's concrete variables. JML's "represents" clause is taken directly from their work [16]. The importance of this result for the present paper is that JML's semantics builds on this work, which allows JML to be used not only as a DBC tool, but also as a formal specification and verification language [14, 24].

5 Discussion

In addition to model fields and model methods, JML also has model classes and model interfaces. These specification-only classes and interfaces are a natural extension of other model declarations. However, the JML compiler does not yet generate any bytecode for model classes or model interfaces, although we hope to implement these in future work. At present, the JML compiler treats any reference to a model type as non-executable.

In general, JML takes the approach of allowing a specifier to use an unrestricted syntax with some non-executable features, and its runtime assertion checker only executes a subset of the language, albeit a large subset. An example of such a non-executable construct is a reference to a model field for which there is no represents clause. Non-executable expressions are handled in a systematic way by JML's contextual interpretation of expressions [5, Chapter 3], which also deals with undefinedness resulting from exceptions that may occur during expression evaluation [14]. The contextual interpretation ensures that such non-executable constructs do not lead to spurious assertion violations.

Our approach to evaluating assertions written with model fields has a shortcoming in terms of performance. Each reference to a model field results in the construction of a new abstract value for the model field. For model fields of class types, this may incur a runtime performance penalty in terms of memory and speed. For example, in our sorted list example (see Figure 5 on page 9), each reference to the model field `theList` creates a new `JMLEqualsSequence` from the program fields `isEmpty`, `val`, `left`, and `right`. Frequent construction of abstract values from concrete representations may adversely affect the speed of runtime assertion checking, especially for large container-style data structures.

Additional costs associated with recomputing abstract values could be reduced in principle by caching. These costs exist primarily as a result of choosing the simplest generation strategy for the current implementation. Model field values need only be generated once within a given assertion (or a sequence of assertions checked adjacently), since evaluation of assertions cannot change the state of the underlying object in JML [14] (and hence cannot change the state stored in model variables, once these values have been computed). We are exploring caching strategies that allow the runtime checker to avoid redundant construction of abstract values.

6 Conclusions

The use of abstract, specification-only features is an evolutionary advance over DBC approaches that use query methods and immutable types. In the query method approach, specifiers add side-effect free methods to be used in specifications, but it is difficult to distinguish methods intended only for use in specifications. Thus program code might use added query methods, and they may take up memory at runtime. Similarly, in the immutable type approach, the program maintains state for specification purposes, but there is no easy way to avoid maintaining this state when assertion checking is disabled. We have explained how these problems are solved in JML by distinguishing model fields and model methods from program fields and program methods.

A similar distinction between model and program entities could be added to other DBC notations and tools. Tools can be built to recognize this distinction and keep the two kinds of declarations separate. Other languages could also adopt notations such as `spec_public` and `pure`, to mark declarations for use in both specification contexts and program contexts.

Together, the ideas described in this paper subsume and extend DBC: everything that can be specified in a language like Eiffel can be written in JML, whether one uses the helper variable approach, the query method approach, the immutable type approach, or some combination. In return for marking the purposes of various declarations, there is less confusion between specification-only code and code intended for use by clients, and the JML compiler can produce more efficient code when assertion checks are turned off. Furthermore, it is easier to write more abstract specifications, since in production code there is no longer any runtime cost associated with using the immutable type approach. Ultimately, model-oriented specifications for interfaces need model fields and methods.

An implementation of the JML compiler along with other JML tools is available from the JML home page at `http://www.jmlspecs.org`.

Acknowledgements

The work of both Cheon and Leavens was supported in part by the US National Science Foundation under grants CCR-0097907. In addition, the work of all authors was supported in part by US NSF grant CCR-0113181. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF. Thanks to David Cok and Curtis Clifton for comments on earlier drafts of this paper.

References

[1] Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, November 2001.

[2] Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, March 2003.

[3] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001. Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001. Available from `www.elsevier.nl`.

[4] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[5] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[6] Stephen H. Edwards, Gulam Shakir, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth. A framework for detecting interface violations in component-based software. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 46–55. IEEE Computer Society Press, June 1998.

[7] J. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[8] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.

[9] W. A. Hegazy. *The Requirements of Testing a Class of Reusable Software Modules.* PhD thesis, Ohio State University, 1989.

[10] C. A. R. Hoare. Notes on data structuring. In E. Dijkstra Ole-J. Dahl and C. A. R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, Inc., New York, NY, 1972.

[11] Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[12] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[14] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 262–284. Springer-Verlag, Berlin, 2003.

[15] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[16] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[17] Barbara Liskov and John Guttag. *Program Development in Java.* The MIT Press, Cambridge, Mass., 2001.

[18] David Luckham. *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1990.

[19] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[20] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[21] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.

[22] Richard Mitchell and Jim McKim. *Design by Contract by Example.* Addison-Wesley, Indianapolis, IN, 2002.

[23] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, October 1994.

[24] Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[25] Bernd Schoeller. Strengthening Eiffel contracts using models. In *Proceedings of the Workshop on Formal Aspects of Component Software (FACS 2003)*, pages 143–158, September 2003. UNU/IIST Report No. 284.

[26] Murali Sitaraman and Bruce W. Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–22, Oct 1994.